Parallel algorithms for identifying convex and non-convex basis polygons in an image

Arijit Laha [a], Amitava Sen [b], Bhabani P. Sinha [b,*]

[a] *Institute for Development and Research in Banking Technology Castle Hills, Masab Tank, Hyderabad 500 057, India*

[b] *Advanced Computing and Microelectronics Unit, Indian Statistical Institute, 203 B.T. Road, Calcutta 700 108, India*

Abstract

In this paper, we propose two novel parallel algorithms for identifying all the basis polygons in an image formed by n straight line segments each of which is represented by its two end points. The first algorithm is designed to tackle the simple situation where all basis polygons are convex. The second one deals with the general situation when the basis polygons can be both convex and non-convex. These algorithms are based on an idea of traversal along the periphery of the basis polygons in a well-defined manner so that each of these needs only $O(n)$ time using an $n \times n$ processor array. Simulation results on various test input sets of intersecting line segments have also been found satisfactory.

Keywords: Parallel algorithm; Basis polygon; Edge traversal

☆ A preliminary version of this work appeared in the Proceedings of the 9th International Conference on High Performance Computing, Bangalore, India, 2002.

* Corresponding author.

E-mail addresses: alaha@idrbt.ac.in (A. Laha), amitavasen@hotmail.com (A. Sen), bhabani@isical.ac.in (B.P. Sinha).

1. Introduction

Finding a higher level description of an image in terms of its constituent objects is a fundamental problem in automatic pattern recognition and computer vision. Preparation of such a description, in general, involves extraction of various information from the image. The field of research that embodies study of such techniques is called image segmentation. In this correspondence we deal with two-dimensional images consisting of only the straight line segments. The fundamental problem associated with such images is that of detecting the straight lines. Most often this problem is tackled by using Hough transform [1,2]. In recent years, a number of parallel algorithms for computing the Hough transform on different architectures have been presented by several authors [3–5]. Asano et al. [6] proposed a scheme for detection of straight lines in an image based on topological walk on an arrangement of sinusoidal curves defined by Hough transform. In [7], a novel parallel algorithm for identifying all straight line segments have been proposed. This algorithm was shown to be capable of overcoming several difficulties associated with the approach using the traditional Hough transform.

However, once the straight line segments in such an image are detected, an even higher level description of the image can be generated in terms of the polygons created by the constituent line segments. Such a description is more useful for syntactic pattern recognition and computer vision tasks. A number of interesting operations on polygons have been investigated by the researchers in computational geometry and computer graphics, once these polygons are identified. These include: (i) finding the convex hull of a polygon [8], [9], (ii) testing the convexity of a polygon [10], (iii) finding the intersection of two convex polygons [11–13], (iv) finding the minimum vertex distance between two crossing convex polygons [14], (v) triangulation of polygons [15,16], etc.

Given the set of straight lines in a two-dimensional image, the set of polygons formed by all these lines can best be described by the *basis polygons*, where a basis polygon is defined as one which would not enclose any other polygon. The maximum number of basis polygons that can be created by n intersecting straight line segments can be estimated as follows.

We start with an empty collection of line segments and add one line at a time to the collection and count the number of new basis polygons that can be created by the line. Let us denote the current number of lines in the collection by k. Since a triangle is a minimal polygon, the first polygon can appear when $k = 3$. When the $(k + 1)$th line is added to the collection of k lines, it can intersect all the pre-existing k lines in a certain order. This $(k + 1)$th line can generate a new basis polygon with two corner points as its points of intersection with a pair of consecutively placed pre-existing lines. So the maximum number of new basis polygons created by the $(k + 1)$th line is same as the number of distinct consecutive pairs of lines that can be found in the collection of k lines, i.e., $k - 1$. Hence, the maximum number of basis polygons created by a collection of n straight line segments is

$$\sum_{k=2}^{n-1}(k-1) = \binom{n-1}{2} \approx \mathrm{O}(n^2)$$

Again, one polygon can have a maximum of n sides. Hence the worst case lower bound of computational complexity for identifying all polygons is $\mathrm{O}(n^3)$.

In this paper, we present two parallel algorithms for detecting all the basis polygons created by n intersecting straight line segments using $n \times n$ array of processors. Both these algorithms need $\mathrm{O}(n)$ time to identify all the basis polygons using $\mathrm{O}(n^2)$ processors in a CREW PRAM model. Thus, in the sense of worst case scenario, the proposed algorithms are optimal. The first one of the two algorithms presented here is designed under the assumption that all the basis polygons are convex (Fig. 1(a)) and no degenerate structures (e.g., a line ending inside a polygon (Fig. 1(b))) or nested polygons (in which one polygon is completely enclosed by another) exists. In Fig. 1(b), the polygon *abebcda* contains a degenerate structure *beb*. The second algorithm is designed to tackle the difficulties introduced by non-convexity, degenerate structures and nested polygons.

2. Basic concepts

In this section we describe the computing model, the notations and terminologies used in developing the algorithms.

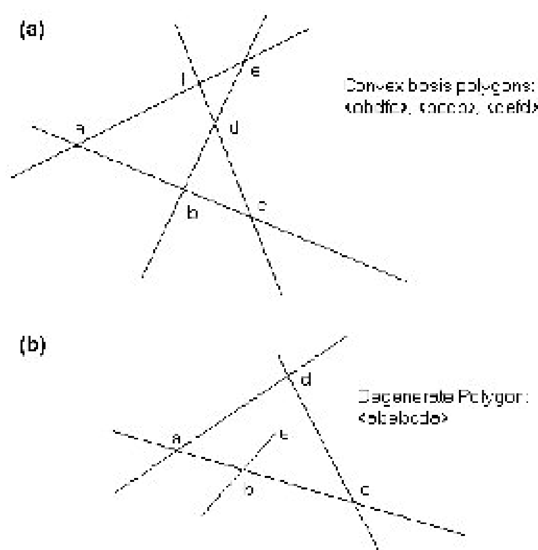

Fig. 1. Example of (a) convex basis polygons and (b) polygon with degenerate structures.

2.1. The computing model

The proposed computing model consists of n^2 processors using a shared RAM in CREW fashion. Each processor will be assumed to have also some local memory. For ease of our later discussions, we would visualize that the n^2 processors are arranged in the form of an $n \times n$ array so that P_{ij} would denote the processor in the ith row and jth column of the array.

2.2. Basic concepts

We assume that the two-dimensional image consisting of intersecting straight lines only, lies in the plane of the paper. We use a two-dimensional orthogonal coordinate system with origin at the lower left corner of the image. The X-axis extends horizontally rightward and the Y-axis extends vertically upward in the plane of the paper.

A point $p = (x, y)$ is denoted by a usual two-tuple of x and y coordinate values.

Definition 1. An ordering among the points on the 2D plane is defined as follows:

(1) a point $p_1 = (x_1, y_1)$ is equal to another point $p_2 = (x_2, y_2)$ if $x_1 = x_2$ and $y_1 = y_2$,
(2) p_1 is on the left of p_2 (denoted as $p_1 < p_2$), if $x_1 < x_2$ or ($x_1 = x_2$ and $y_1 < y_2$),
(3) p_1 is on the right of p_2 (denoted as $p_1 > p_2$), if $x_1 > x_2$ or ($x_1 = x_2$ and $y_1 > y_2$).

Example 1. In Fig. 1(a) the point a is the leftmost one. So we can write $a < b$, $a < c$, etc. Similarly other points in Fig. 1(a) can be related as $c > d$, $d < e$, etc. However, if two points are such that the line segment joining them is parallel to Y-axis (such points are not shown in Fig. 1(a) and (b)), e.g., $p_1 = (20, 20)$ and $p_2 = (20, 30)$ then according to our definition, $p_1 < p_2$.

A line from the point p_1 to p_2 is denoted by $\vec{p_1p_2}$.

Definition 2. A point p_3 is said to be on the *left-hand side* of the line $\vec{p_1p_2}$ if the cross product $\vec{p_1p_2} \times \vec{p_1p_3}$ points perpendicular to the image plane and towards the viewer, i.e., $(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1) > 0$.

Example 2. In Fig. 1(a), points d, e and f are on the left-hand side of the line $\vec{ab}$ while the point c is on the right hand side of line $\vec{bd}$.

In our algorithms we shall often have occasions for computing the angle between two lines. The angle θ between two lines $\vec{p_1p_2}$ and $\vec{p_1p_3}$ is computed from the dot product of the vectors as $\theta = \cos^{-1}\vec{p_1p_2} \cdot \vec{p_1p_3} / (|\vec{p_1p_2}||\vec{p_1p_3}|)$.

The input to our algorithms will be a set of n lines $\{L_i : i = 1, 2, \ldots, n\}$, each line being expressed as an ordered 2-tuple of end points (p_l, p_r), where p_l is the left end point and p_r is the right end point, i.e., $p_l < p_r$.

To facilitate our following discussions, whenever we would refer to a point, this would exclusively mean either an end point of a straight line or an intersection point of a set of straight lines.

Definition 3. A point p_i said to be a neighbor of another point p_j on a line $\bar{L}_k$ if (1) $p_i \neq p_j$ and (2) there is no other point on $\bar{L}_k$ between p_i and p_j.

Remark. An intersection point can have one neighbor (if it is an end point of $\bar{L}_k$ as well) or two neighbors on a line; an end point can have only one neighbor on a line.

If a neighboring point is on the left side of a point, then it is called a left neighbor of the point, otherwise it is called a right neighbor.

Example 3. In Fig. 2, four lines (L_1, L_2, L_3 and L_4) with end points as (l_1,r_1), (l_2,r_2), (l_3,r_3) and (l_4,r_4), respectively, are shown. The point a has the points l_1 and b as its left and right neighbors, respectively on line 1. Point a is also on line 2 and it has points l_2 and c as left and right neighbors, respectively on line 2. However, l_1 has only a as its right neighbor.

The intersection point of lines $\bar{L}_i$ and $\bar{L}_j$ can be referred to in two possible ways: (1) as an intersection point lying on the line $\bar{L}_i$, and also (2) as an intersection point lying on the line $\bar{L}_j$. More than two lines may intersect at the same point. Such intersection points will also be referred to using the indices of any two intersecting lines.

Example 4. In Fig. 2, a is the point of intersection of lines L_1 and L_2. It can be referred to as either *Intersection*(1,2) or *Intersection*(2,1). Again b is the point of intersection of lines L_1, L_3, and L_4, and it can be referred to by anyone of *Intersection*(1, 3), *Intersection*(3,1), *Intersection*(1,4), *Intersection*(4,1), *Intersection*(3,4) and *Intersection*(4, 3). This flexibility of referring to the same intersection point in different ways will be used in describing our proposed algorithms.

A polygon will be represented by a list of vertices $\{v_i: i = 1, 2, \ldots, s\}$ satisfying the following conditions: (1) $v_1 = v_s$, (2) each vertex v_i is either an intersection point of two or more lines or a pure endpoint (such vertices may appear if degenerate

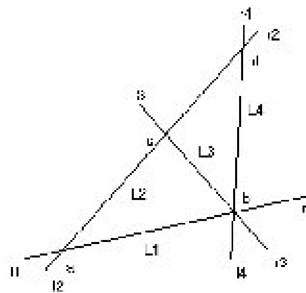

Fig. 2. An image with four lines forming two basis polygons.

structures (Fig. 1(b)) exist), (3) any pair of consecutive vertices v_i and v_{i+1} are joined by a single line segment and (4) there is no other intersection point between v_i and v_{i+1} (if so, that point should have appeared between v_i and v_{i+1} in the vertex list).

Example 5. Fig. 1(a) contains three basis polygons described by vertex lists <*abdfa*>, <*bcdb*> and <*dafd*>. Fig. 1(b) depicts a polygon with degenerate structures which is represented by the vertex list <*abebcda*>.

The convex polygons as shown in Fig. 1(a) and Fig. 2 are the simplest to work with. They are convex, without degenerate or nested structures and if two polygons are adjacent then they share one and only one common edge. However, in our proposed algorithms, we would like to consider the detection of basis polygons for the following polygonal structures:

(1) A vertex of a polygon may be on an edge of another polygon, but none is enclosed by the other. We call these polygons as *touching polygons*.

Example 6. In Fig. 3(a), polygons <*abefa*> and <*bcdb*> are two adjacent polygons sharing only the point *b*, and these are called touching polygons.

(2) A polygon may be *non-convex* (Fig. 3(b)).
(3) One or more lines having their end point(s) within a polygon. We call them *polygons with degenerate structure*.

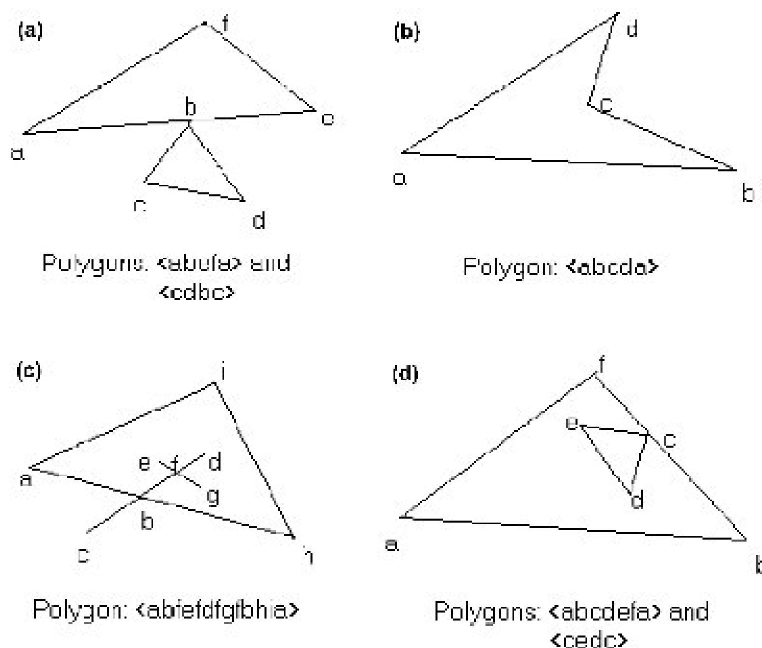

Fig. 3. The situations which can be faced in general cases: (a) touching polygon; (b) non-convex polygon; (c) degenerate structure and (d) nested polygon.

Example 7. In Fig. 3(c), the polygon <*abfefdfgfbhia*> has one end point of line $\vec{cd}$ and a whole line $\vec{eg}$ within it.

(4) One polygon may be contained within another polygon and a vertex of the inner polygon is on the periphery of the outer polygon (Fig. 3(d)). We call the inner one to be *nested* within the outer one.

Example 8. In Fig. 3(d), the polygon <*cdec*> is contained within a outer polygon <*abcfa*>. In terms of basis polygon they should be detected as polygons <*abcdecfa*> and <*cdec*>.

Definition 4. A polygon with any combination of four situations listed above will be called a *general polygon*.

3. Algorithms

The algorithms presented in this section are based on ordered traversals along the sides of the potential basis polygons. Each traversal starts from an intersection point on a line, proceeds along this line towards the right neighbor of the point on the line. At that point the traversal chooses another neighboring point as the next vertex. To do this the traversal has to find out all the neighboring points of the current point and choose one of them. The criteria which determine the choice of the next vertex to move on, will be properly formulated to ensure that the traversal proceeds along the sides of a potential basis polygon. If the traversal returns to the starting point, a basis polygon is detected and the list of the points visited by the traversal in the order of the visit, forms the vertex list of the corresponding basis polygon. If the traversal fails to find the next vertex, it is aborted.

In the algorithms developed here, each intersection point has a special correspondence to a particular processor, i.e., the intersection point *Intersection*(i,j) of lines i and j has a special correspondence to the processor P_{ij}. The processor P_{ij} would initiate a traversal from the *Intersection*(i,j) along the line i. In the first algorithm dealing with detection of simple convex polygons, one processor traverses one basis polygon and produces the corresponding vertex list. In the second algorithm more than one processor can traverse parts of a basis polygon and each of them produces a partial vertex list. At the end of traversal these partial vertex lists are collected and concatenated in the proper order by only one of the processors involved, so that the whole vertex list corresponding to a basis polygon is produced. This feature simplifies the detection of complicated situations described above, utilizes available computing resource more effectively and reduces the overall computation time. The details of the algorithms are presented in following subsections. However, in the next example we briefly introduce the general strategy followed in the algorithms. This example deals with convex polygons only.

Example 9. In Fig. 2, the processor P_{12} starts a traversal from a along line L_1 towards its neighboring intersection point b (in rightward direction). After reaching

b, all the neighboring points on all lines other than L_1 (the current line) intersecting at b (current point) are considered. These are: c and r_3 on line L_3, d and l_4 on line L_4. Out of these, we would choose one point to traverse next such that a sharpest possible left turn can be made. The point c satisfies this requirement for this example. The traversal proceeds to c and using the same strategy, a is chosen as the next vertex. Since a was the starting vertex, this completes the traversal and the basis polygon <*abca*> is detected. On the other hand another traversal initiated by the processor P_{21} from a along line L_2 is aborted when it reaches l_3 via c.

3.1. Algorithm for convex polygons

We first describe a parallel algorithm in order to detect the basis polygons for the special case when these are all convex, and there is no degenerate or nested structure. Through this simplification we can focus on the key aspects of the algorithm and treat the more generalized algorithm as an extension of the simpler one. The algorithm takes the set of straight lines as the input; each line being represented by its two end points. From this input it first constructs four matrix data structures as follows and then uses them to detect the polygons.

3.1.1. Data structures

(1) *InterSect*: An $n \times n$ matrix for storing intersection points. $InterSect_{ij}$ stores the intersection point of lines $\bar{L}_i$ and $\bar{L}_j$ or an *invalid* value if $\bar{L}_i$ and $\bar{L}_j$ do not intersect. All elements are initialized as invalid.

Remark. *InterSect* is a symmetric matrix with diagonal terms having invalid values (since $\bar{L}_i$ cannot intersect itself).

Example 10. The data structures corresponding to the image in Fig. 2 are shown in Fig. 4. The first row in the matrix *InterSect* contains the intersection points on line 1, *Invalid*, a, b and b, respectively. Note that b is the intersection point of lines L_1, L_3 and L_4, and hence, it has been entered in both $Intersect_{13}$ and $Intersect_{14}$.

(2) *SortedInterSect*: An $n \times (n + 1)$ matrix for storing sorted intersection points on each line (as appearing in the matrix *InterSect*) and end points. Suppose a line $\bar{L}_i$ intersects with k, $0 \leqslant k \leqslant n - 1$ lines. Then the corresponding k intersection points (all of them may not be distinct) are placed in columns 2 through $k + 1$ of the ith row of *SortedInterSect*, in order of non-decreasing rightwardness. Thus, the entries in columns 2 to $k + 1$ of row i of *SortedInterSect* are generated by sorting the non-diagonal entries in row i of the matrix *Intersect*. Column 1 of each row of *SortedInterSect* will contain the left end point (if it is not also an intersection point), or an invalid entry (if the left end point is also an intersection point). Column $k + 2$ contains the right end point if it is not also an intersection point; otherwise it contains an invalid entry. All columns from $k + 3$ to $n + 1$ contain invalid entries.

Remark. The neighboring intersection points (on the left or on the right) of a given intersection point on a line can be easily obtained from the matrix $SortedInterSect_{ij}$.

InterSect

Invalid	a	b	b
a	Invalid	c	d
b	c	Invalid	b
b	d	b	Invalid

SortedInterSect

l1	a	b	b	r1
l2	a	c	d	r2
l3	c	b	b	r3
l4	b	b	d	r4

InterSectIndex

-1	2	3	4
2	-1	3	4
3	2	-1	4
3	4	3	-1

SortingIndex

-1	2	3	4	-1
-1	1	3	4	-1
-1	2	1	4	1
-1	1	3	2	-1

Fig. 4. Data structures associated with the image shown in Fig. 2.

Example 11. In Fig. 4, the first row of the matrix *SortedInterSect* contains the entries l_1,a,b,b and r_1 respectively. They represent the following facts: (i) l_1 is the left end point of line 1 and l_1 is not an intersection point. (ii) a, b and b are three intersection points on line L_1 arranged in the order of non-decreasing rightwardness. Third and fourth entries are identical, so there are two lines intersecting with line L_1 at b. However, the line number corresponding to the third entry is less than that of the fourth entry. (iii) r_1 is the right end point of line L_1 and r_1 is not an intersection point.

(3) *SortingIndex*: An $n \times (n + 1)$ matrix of integer values. Its purpose is to create a positional correspondence between the sorted entries of the intersection points in *SortedInterSect* and their original (assorted) entries in *InterSect*. Thus, if $SortedInterSect_{ij}$=$InterSect_{i,k}$, then $SortingIndex_{ij} = k$. For non-distinct entries in *SortingIndex*, if $SortedInterSect_{ij}$=$SortedInterSect_{i,j+1}$, then the corresponding entries in $SortingIndex_{ij}$ and $SortingIndex_{i,j+1}$ are filled up with index values $k1$ and $k2$ (when $SortedInterSect_{ij} = SortedInterSect_{i,j+1} = InterSect_{i,k1} = InterSect_{i,k2}$), such that $k1 < k2 < \ldots$. All entries in *SortingIndex* corresponding to pure end points and invalid values in *SortedInterSect* matrix are set to an invalid index value of -1.

Example 12. Again we use the first row of *SortingIndex* in Fig. 4 as the example. The entries are -1, 2, 3, 4 and -1 respectively. They signify that (a) there are no entries corresponding to l_1 and r_1 in the first row of the matrices *InterSect* and *InterSectIndex* (since they are pure end points) and (b) the entries corresponding to the intersection points a, b and b in first row of *SortedInterSect* can be found in the columns 2, 3 and 4 respectively in the first row of the matrices *InterSect* and *InterSectIndex*. Thus this matrix can be used to find entries in the matrices *InterSect* and *InterSectIndex* corresponding to a point whose position in the matrix *SortedInterSect* is known.

(4) *InterSectIndex*: An $n \times n$ matrix of integer values. Its purpose is to create an inverse correspondence as done by *SortingIndex*. Thus, if $InterSect_{ij} = SortedInterSect_{ik}$ then $InterSectIndex_{ij} = k$. All entries corresponding to invalid values in *InterSect* matrix are set to an invalid index value of −1. In case of non-distinct entries in *SortedInterSect*, the corresponding entries in *InterSectIndex* are so entered that the index values are in ascending order from left to right.

Example 13. Again we refer to the first row of *InterSectIndex* in Fig. 4 as the example. The entries −1, 2, 3 and 4 tell us that the entries corresponding to the intersection points $InterSect_{12}$,$InterSect_{13}$ and $InterSect_{14}$ can be found in the columns 2, 3 and 4 respectively in the first row of the matrix *SortedInterSect*.

Before presenting the algorithm in a formal manner we first describe its basic idea with the help of the example of Fig. 2.

Each basis polygon will be outputted by means of a list of its vertices in order, with the first and last vertices being the same. For example, the two basis polygons in Fig. 2 may be outputted as <*abca*> and <*bdcb*>. The essential idea of our proposed algorithm is to generate the vertex list for each basis polygon by an appropriate traversal along the edges of the polygons. A traversal for detecting a basis polygon starts from an intersection point on one of the intersecting lines. This point is marked as the *starting vertex* and put in the vertex list describing a basis polygon. The line on which this starting vertex lies, is termed (temporarily) as the *current line*. Suppose we start from the point a on line L_1, i.e., a is the starting vertex, and line L_1 is the current line. The next vertex is always the closest intersection point on the current line on the right side of the starting point. This is called *Start Rightward* strategy. The next vertex can be found in the following way. Since a is the intersection point of lines L_1 and L_2, $InterSect_{12} = InterSect_{21} = a$. Starting from $InterSect_{12} = a$, we move along the current line, i.e., line L_1. The right neighbor of a on line L_1 can be found from row 1 of *SortedInterSect*, if we know the position of the entry corresponding to the intersection point a ($=InterSect_{12}$) in this row. This is done by looking up the $(1,2)$th entry in the matrix *InterSectIndex*, which gives the value 2. Thus, column 2 in row 1 of *SortedInterSect*, i.e., $SortedInterSect_{12}$ is the entry corresponding to $InterSect_{12}$. Hence, $SortedInterSect_{13} = b$ is the next vertex on right of a along line L_1, and b is added to the vertex list of the polygon.

To continue the traversal from b, we need to (1) first identify all the lines (other than the current line) intersecting at b, (2) find the two neighboring points of b on each of these intersecting lines (a neighboring point may also be an end point), and (3) finally select one of all these neighboring points for inclusion in the vertex list representing the basis polygon. Note that we treat both the intersection points and end points as possible candidate points for inclusion in the vertex list at this stage. The differentiation between these two types of points will be done later.

For the first step, we first make a rightward scan on row 1 of *SortedInterSect* starting from its $(1,2)$th entry to get $SortedInterSect_{13} = SortedInterSect_{14} = b$. The identity of the lines intersecting with line L_1 at b can be discovered by looking up the entries in *SortingIndex* corresponding to $SortedInterSect_{13}$ and $SortedInterSect_{14}$.

We find that $SortingIndex_{13} = 3$ and $SortingIndex_{14} = 4$. Thus, the lines intersecting with line L_1 at b are lines L_3 and L_4.

For the second step, we have to find out the neighbors of b on lines L_3 and L_4. For this, we first identify the entries in *SortedInterSect* corresponding to $InterSect_{31}$ and $InterSect_{41}$. Now $InterSectIndex_{31} = 3$ and $InterSectIndex_{41} = 2$. Thus, $SortedInterSect_{33}$ and $SortedInterSect_{42}$ are the required entries corresponding to $InterSect_{31}$ and $InterSect_{41}$, respectively. Looking up the neighboring columns of elements $(3, 3)$ and $(4, 2)$ in *SortedInterSect*, we find two candidate points $SortedInterSect_{32} = c$ and $SortedInterSect_{35} = r_3$ on line L_3, and two other candidate points $SortedInterSect_{41} = l_4$ and $SortedInterSect_{44} = d$ on line L_4.

For the third step, to choose the point to move from among the candidate points, a two stage strategy is used. The traversal will move to a point if the point is on the left of the current line. This is called *Move Left* strategy. Each of the intersecting lines in consideration can contribute at most one candidate point satisfying this condition. In our case c and d are those candidate points. Among these points, the one on the line that makes smallest angle with the current line on the left side is chosen. This is known as *Move Left Sharpest* strategy. In our case $\angle abc < \angle abd$. So c is chosen as the next vertex and added to the vertex list and line L_3 is set as current line.

Following in a similar manner, line L_2 can be found as intersecting line L_3 at point c, and hence, the point a on line L_2 will s polygon, according to the *move left strategy*. Since a is the start vertex, the algorithm ends successfully detecting the basis polygon *abca*.

Similarly, a traversal starting from b along line L_4 would detect the polygon *bdcb* and a traversal starting from a along line L_2 will end without success when it reaches l_3 via c.

The formal presentation of the algorithm is shown in Fig. 5. The following notations have been used in describing the algorithms:

1. Operations on a collection of data elements (e.g., list, array) has been denoted as *collection_name.operation(arguments)*.
2. Some functions compute a tuple of two values. These will be represented in the form $[Value1, Value2] \leftarrow function_name(arguments)$.

The functions *Intersection*, *LeftEnd* and *RightEnd* in Fig. 5 are self-explanatory.

The procedure *DetectPolygon* is described in Fig. 6. In this procedure, data type *Point* is a 2-tuple of co-ordinate values, type *Identity* is a 2-tuple of index values (for processors) and type *LineNumber* is an integer identifying a line. The functions and several termination conditions used in DetectPolygon are explained below:

The variable *Position* is a 2-tuple of index values corresponding to the entry of the *NextVertex* in *SortedIntersect*. The function *RowValue*(*Position*) extracts the row index part, which is the number of the line connecting the *CurrentVertex* and the *NextVertex* that becomes the *CurrentLine* for the next step of traversal. The column index for the entry of *NextVertex* in the *CurrentLine*-th row of *SortedInterSect* is extracted with the function *ColumnValue*(*Position*).

Algorithm:

Step 1: For $i, j = 1$ to n and $i \neq j$

All processors P_{ij} do in parallel

$InterSect_{ij} \leftarrow$ **Intersection**(L_i, L_j);

Step 2: For $i = 1$ to n

Each row of processors P_i do in parallel

$[SortedInterSect_{i,2:n+1}, SortingIndex_{i,2:n+1}] \leftarrow$ **ParallelSort**$(InterSect_i)$;

/* In the above step, the processors in the ith row sort (in parallel) the intersection points in the ith row of the matrix *InterSect* to produce the sorted intersection points in the ith row of the matrix *SortedInterSect*, filling its columns 2 through $n + 1$. The processors also fill the corresponding entries in the matrix *SortingIndex* with the column numbers that the sorted entries originally occupied in the matrix *InterSect*. In the above sorting procedure it was assumed that $Invalid > a$, for any valid point a. */

Step 3: For $i = 1$ to n

All processors P_{ii} do in parallel

If LeftEnd$(L_i) \neq SortedInterSect_{i,2}$

$SortedInterSect_{i,2} =$ **LeftEnd**(L_i);

$SortedInterSect_{i,1} = Invalid$;

Else

$SortedInterSect_{i,1} =$ **LeftEnd**(L_i);

$j \leftarrow 2$;

While $(j \leq n + 1)$**AND**$(SortedInterSect_{i,j} \neq Invalid)$ do

$InterSectIndex_{i,SortingIndex_{ij}} \leftarrow j$;

$j \leftarrow j + 1$;

Endwhile;

If RightEnd$(L_i) \neq SortedInterSect_{i,j-1}$

$SortedInterSect_{i,j} \leftarrow$ **RightEnd**(L_i);

/* In this step one processor from each row of processors (we choose the processor P_{ii} from i^{th} row) works on the corresponding row of the matrix *SortedIntersect* to deal with the possibility that the end points also can be intersection points. Each processor also prepares the corresponding row of the matrix *InterSectIndex*. */

Step 4: For $i, j = 1$ to n and $i \neq j$

All processors P_{ij} do in parallel

If $InterSect_{ij} \neq Invalid$

DetectPolygon(i, j);

/* In this step each processor corresponding to each valid intersection point starts (in parallel) executing the procedure for detecting a polygon. */

Fig. 5. Algorithm for basis polygon detection.

```
Procedure DetectPolygon(i, j)
List VertexList; /* List of vertices. */
List CandidateLines; /* List of line numbers. */
List CandidatePoints; /* List of points. */
List PointPositions; /* List of positions (two-tuples of indexes). */
Point StartVertex, CurrentVertex, NextVertex;
Identity Position;
LineNumber CurrentLine;

VertexList ← Empty;
StartVertex ← InterSect_{i,j};
VertexList.add(StartVertex);
    /* The above operation represents adding a vertex to the vertex list. */
CurrentLine ← i;

k ← InterSectIndex_{ij};
If SortedInterSect_{i,k+1} = StartVertex
    Terminate^1;

CurrentVertex ← SortedInterSect_{i,k};
VertexList.add(CurrentVertex);
SortedColumn ← k;

While (CurrentVertex ≠ StartVertex) do
    CandidateLines ← FindCandidateLines(CurrentLine, SortedColumn);
    If CandidateLines is empty
        Terminate; /* CurrentVertex is a pure end point. */

    [CandidatePoints, PointPositions] ← FindCandidatePoints(CandidateLines, CurrentLine);
    [NextVertex, Position] ← FindNextVertex(CandidatePoints, PointPositions);
    If NextVertex = Invalid
        Terminate; /* No left move is possible. The traversal has to end. */
    If NextVertex < StartVertex
        Terminate^2;

    CurrentLine ← RowValue(Position);
    SortedColumn ← ColumnValue(Position);
    CurrentVertex ← NextVertex;

    VertexList.add(CurrentVertex);
End While.

Report polygon.

End DetectPolygon
```

Fig. 6. The DetectPolygon procedure for convex basis polygons.

Terminate[1]: If the *StartVertex* is an intersection of more than two lines, more than one processor will start from the same point along the same line, thus performing duplicate computation. The condition Terminate[1] allows only one processor (the processor having the lowest value of column index) to start traversal from a point along a particular line.

The function *FindCandidateLines* uses its arguments to find the *CurrentVertex* in the *SortedInterSect* and scans the neighboring columns for *CurrentVertex*. Then it finds the line identities for all entries of *CurrentVertex* from the *SortingIndex* and outputs the list *CandidateLines* with the values found in *SortingIndex*. Thus, this list contains the number of each line intersecting with *CurrentLine* at the *CurrentVertex*.

The function *FindCandidatePoints* takes the list *CandidateLines* and *CurrentLine* as input, and it outputs the list of two tuples [*CandidatePoints*,*PointPositions*]. This function first looks up the *InterSectIndex* using the values in the list *CandidateLines* as row indexes and *CurrentLine* as the column index. The values found are the column indexes for the entries of *CurrentVertex* in *SortedInterSect* in the rows corresponding to the lines in *CandidateLines*. Then for each of these entries of *CurrentVertex* the left and right neighbors are found in the *SortedInterSect*. Each of these neighbors found is listed in *CandidatePoints*, and the values corresponding to the row and column indexes in *SortedInterSect* are entered in the list *PointPositions*.

The function *FindNextVertex* takes the list of 2-tuples [*CandidatePoints*,*PointPositions*] as input, and it outputs the tuple [*NextVertex*,*Position*] to indicate the point where to move next. It first applies the *Move Left* strategy to select a candidate point. If more than one point qualify for left move, *Move Left Sharpest* strategy is used to select a single point. The selected point is returned as the *NextVertex* and its row and column indexes as found in *PointPositions* are returned as *Position*. If no point qualifies for left move, an invalid value is returned for *NextVertex*.

Terminate[2]: If *NextVertex* < *StartVertex* and both of them are vertices of the same basis polygon, then *StartVertex* must be reachable from an intersection point which is again a part of the same polygon using a right move (initial move), and the vertices computed so far must be the part of the vertex list of the processor starting from that intersection point also. Thus ultimately, at least two processors will report the same polygon, with more than one processor ending up with detecting the same basis polygon. Using this Terminate[2] condition, we allow only that processor to complete the traversal for which the point it is starting from is the *leftmost* vertex of the polygon.

3.1.2. Complexity of the algorithm

In step 1 of the algorithm in Fig. 5, each processor P_{ij} finds the intersection of lines L_i and L_j by simply solving the equation of the lines. This step involves O(1) time. In step 2 processors in each row i works as a linear array of processor to sort the intersection points on the ith line. They also generate sorting indexes for the line. This step can be performed in $O(n)$ time if simple Odd–Even Transposition is used. In step 3 one processor in each row i augments the ith row of *SortedInterSect* with the values of the end points of line L_i (provided they are not intersection points as

well) and computes the ith row of the *InterSectIndex*. This is an O(n) operation. In step 4 all the processors P_{ij} for which $InterSect_{ij}$ is a valid point starts a traversal for detecting a basis polygon with start vertex $InterSect_{ij}$ along line L_i.

If a polygon has $s \leqslant n$ (since polygons are convex) sides the traversal that detects it has to cover s sides. If all the intersection points encountered en route are intersection of two lines, choosing the next vertex at each vertex is an O(1) operation. However if a vertex is intersection of $k > 2$ lines the algorithm has to choose one line out of $k - 1$ lines. Thus finding the next vertex is an O($k - 1$) operation. But due to the convexity of the polygons $k - 2$ lines rejected at one vertex can never be sides of the same polygon. So the total number of computations for finding the next vertex is at most O(n) for a polygon. Since one polygon is detected by one processor, step 4 of the algorithm also has the worst case complexity of O(n). Thus the whole algorithm has the time complexity of O(n). The algorithm uses four data structures each of size n^2, with a space complexity of O(n^2).

3.2. Algorithm 2

We now modify the algorithm in the previous section so as to consider the detection of basis polygons of the most general type that can be produced by the line segments. The modified algorithm uses the same data structures as the former one and its steps of computation are also same as that of Fig. 5. Only the procedure *DetectPolygon* will be modified. We make two changes as explained below.

First, we change our strategy for finding the next vertex. In the previous algorithm the next vertex is detected only if a left move from the current vertex enables the traversal to reach it. Otherwise, the traversal fails to continue. Now the detection of the next vertex is done using the following strategy consisting of several steps:

(1) Search for a possible left move from the current line. If succeeded, then take the vertex corresponding to the left move chosen as the next vertex; otherwise go to the second step below.
(2) Find the next vertex on the current line (i.e, the neighbor of the current vertex on the current line along the direction of traversal). We call it *Continue along the same line* strategy. If such a vertex is found, then take that as the next vertex; otherwise (i.e., the current vertex is an end point of the current line) go to the third step below.
(3) Find a possible *right move* from the current line. If there exist more than one candidate point for the right move (current vertex is the point of intersection of more than two lines), then choose, as the next vertex, the point in that line which makes the biggest angle with the current line at the current vertex. We call this strategy *Move Right Widest*. If no such right move is possible (i.e., the current vertex is a pure end point of the current line), then go the fourth step below.
(4) Set the vertex prior to the current vertex as the next vertex. We call this *Turn around* strategy.

Another change is concerned with the generation of the vertex list during the traversal of basis polygons. Our goal is to consider the general case of non-convex, degenerate, touching and nested basis polygons without generating duplicate basis polygons. To achieve this, we now propose that each processor P_{ij} corresponding to a valid entry in *InterSect*$_{ij}$ starts a rightward traversal along line L_i as before. However, if at a vertex (say, b) during the traversal, the next vertex (say, c) is the right neighbor of the current vertex b, then the processor P_{ij} stops its traversal creating just a partial vertex list of a basis polygon. The partial vertex list of P_{ij} would then consist of the list of the traversed vertices upto the vertex (say, a) previous to the current vertex (i.e., the current vertex is not included in its partial list). Since c is the right neighbor of b, there must be another processor that has started a traversal along the same route (from b to c) to traverse the same basis polygon as that by P_{ij}. Let this processor be P_{kl} which can easily be detected by the processor P_{ij} by noting the line on which b and c lie. We call the processor P_{kl} as the successor of P_{ij} in this traversal. Conversely, P_{ij} will be termed as the predecessor processor of P_{kl}.

The processor P_{ij} at this stage also writes its own index values (i,j) at a location in the shared memory designated as the *Predecessor Index Buffer* of P_{kl}.

Example 14. Fig. 7 depicts six line segments forming a non-convex polygon. Since a processor corresponding to a valid intersection point starts initially a rightward traversal, the processor P_{12} would start traversing along line 1 from the intersection point a and then it finds the vertex b as the current vertex. At b, it chooses a left move and c becomes the next vertex in the traversal. However, since $b < c$ (i.e., c is the right neighbor of b on the line 3), it implies that the processor P_{31} also has started a traversal from b along line 3 towards c. If P_{12} continues the traversal it will detect the same polygon as the one detected by P_{31}. So instead of proceeding further, P_{12} stops its traversal and keeps the partial vertex list computed by it before the current vertex (b here), i.e., the partial vertex list for P_{12} is $<a>$ and it writes its identity (1, 2) in the predecessor index buffer of the processor P_{31}.

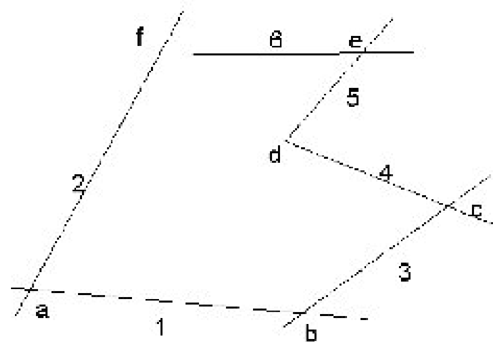

Fig. 7. Six line segments forming a non-convex polygon.

Now, let us follow the traversal initiated by P_{31}. This traversal moves to d via c. Here it finds e as the next vertex (a right move). Since $d < e$, it implies that P_{54} has started a traversal from d towards e along line 5. So P_{31} identifies P_{54} as its successor, stops the traversal with a partial vertex list <*bc*> and writes its own identity (3, 1) in the predecessor index buffer of P_{54}. Similarly, the processor P_{54} traverses the points d, e, f, and a. At a it chooses the next vertex b and finds that $a < b$. Thus, it detects P_{12} as its successor, stops with its partial vertex list <*def*> and writes its identity (5,4) in the predecessor index buffer of $P(1,2)$. The lists and the predecessor index buffers of different processors are as follows:

P_{12}: Vertex list = <*a*>, predecessor index = (5, 4);
P_{31}: Vertex list = <*bc*>, predecessor index = (1, 2);
P_{54}: Vertex list = <*def*>, predecessor index = (3, 1).

We thus see that by this process, we would have a chain of processors for a basis polygon each of them holding a partial vertex list for the complete basis polygon in a definite order. Since a polygon cannot have more than n vertices, it follows that the above process will be completed in O(n) parallel time. Now the remaining task is to collect and merge these partial vertex lists to produce the full vertex list forming a basis polygon. For that we need to devise a protocol for selecting one processor in the chain that will compile the full vertex list of the respective polygon.

We fix our idea that the processor with the lowest index value forming a chain of such partial vertex lists will collect all these partial lists and form the complete vertex list representing the basis polygon in consideration. At this stage every processor will start reading its own predecessor index buffer, follow the index value and read the predecessor index values of the successive processors in the chain, until it reads its own index value. By this, every processor in a chain forms an index list of all processors holding the partial vertex lists of a specific basis polygon. This requires O(n) parallel time. Every processor will then find in parallel the minimum of all these (maximum n in number) index values in log$n$ time.

Only the processor whose index is equal to this minimum index found by the above step, will continue with merging the partial vertex lists of the processors in the chain, as formed by the processor index buffer values of the successive processors. All other processors will exit. This requires O(n) time again.

Example 15. The minimum of the index values for the above Example 14 is (1, 2). Hence, only P_{12} will continue to form the complete vertex list and the others will stop. P_{12} first takes its own list <*a*>, concatenates it after the list <*def*> of its predecessor processor P_{54} to form <*defa*>. Then it concatenates <*defa*> after the partial vertex list <*bc*> of the predecessor processor P_{31} of P_{54} to form the list <*bcdefa*>. The predecessor processor of P_{31} is P_{12}, and so the concatenation process ends there with the final vertex list as <*abcdefa*>.

We now present the modified version of the DetectPolygon procedure formally in Fig. 8. New functions used are explained below:

In Fig. 8, the variables *OwnIdentity*, *Successor* and *Predecessor* are used for storing the own index value of the executing processor, index values of its successor and predecessor processors, respectively. *temp* is a temporary variable also of type *Identity*. The data type *Matrix* is used to denote two-dimensional ($n \times n$) arrays. *PredecessorList* (shared global) and *IndexList* (local) are two such arrays of type *Matrix* which are used for storing the index values of the predecessor of all processors, and the index values of the processors in a chain which hold the partial vertex lists corresponding to a specific basis polygon, respectively.

GetPreviousVertex returns the vertex previous to *CurrentVertex* as *NextVertex* and prepares the *Position* information from the indexes of the entry corresponding to the *NextVertex* in the same row of *SortedInterSect*. This effectively reverses the direction of traversal along the same line. *FindLeftMove* is just another name of *FindNextVertex* used previously. *FindPointOnLine* finds the next neighbor of *CurrentVertex* on the current line along the current direction of traversal. *FindRightMove* returns a vertex for right move chosen using *Move Right Widest* strategy. The function *FindSuccessor*(*Position*) returns the identity of the processor that starts a traversal from the intersection point indicated by *Position*.

3.2.1. Time complexity

We now consider the time complexity of the modified DetectPolygon as in Fig. 8. Non-convex polygons and degenerate structures can make a line appear in a polygon more than once so that the total number of sides in a polygon may exceed n. However for each reappearance of a line the traversal must choose a right neighbor as the next vertex in between. In this algorithm a processor stops traversal whenever a right move is detected. So the computational complexity for the contribution made by an individual processor never exceeds $O(n)$. Thus the overall time complexity of the modified algorithm is also $O(n)$.

3.2.2. Load balancing and scalability

As mentioned earlier, each processor $P_{ij}, i \neq j$ does the computation associated with the possible intersection point of lines i and j, if any. Thus, the diagonal processors do not take part in any computation. Further, other processors corresponding to non-existent (invalid) intersection points (all non-diagonal processors are engaged when each line is intersected by every other line, which is not a general scenario) also remain largely unutilized. However, for the sake of simplicity of the algorithms we do not attempt any finer load balancing.

When the number of available processors is k^2, rather than n^2, the algorithm can be easily scaled with a simple load balancing scheme. Each processor can be associated with $\lceil\frac{n}{k}\rceil \times \lceil\frac{n}{k}\rceil$ intersection points, i.e., P_{ij} is associated with the intersection points of lines having numbers within the range $(i-1)\lceil\frac{n}{k}\rceil + 1$ to $i\lceil\frac{n}{k}\rceil$ with lines having numbers within the range $(j-1)\lceil\frac{n}{k}\rceil + 1$ to $j\lceil\frac{n}{k}\rceil$. The scheme can be easily understood if we think of an $n \times n$ matrix of the intersection points, where ijth element correspond to intersection of lines i and j, and the $k \times k$ matrix of the available processors. The $n \times n$ matrix is first partitioned into $k \times k$ array of submatrices of size

```
Procedure DetectPolygon(i, j)
Point  StartVertex, CurrentVertex, NextVertex;
LineNumber  CurrentLine;
List  VertexList, CandidateLines, CandidatePoints, PointPositions;
Identity  Position, OwnIdentity, Successor, Predecessor;
Matrix  PredecessorList, IndexList;
VertexList ← Empty;    StartVertex ← InterSect_ij;
VertexList.add(StartVertex);
CurrentLine ← i;
k ← InterSectIndex_ij;
If SortedInterSect_i,k+1 = StartVertex
    Terminate^1;
CurrentVertex ← SortedInterSect_i,k;
VertexList.add(CurrentVertex);
SortedColumn ← k;
Flag ← TRUE;
While (CurrentVertex ≠ StartVertex AND Flag = TRUE) do
  CandidateLines ← FindCandidateLines(CurrentLine, SortedColumn);
    If CandidateLines is not empty      /* CurrentVertex is a pure end point*/
      [CandidatePoints, PointPositions] ← FindCandidatePoints(CandidateLines, CurrentLine);
      [NextVertex, Position] ← FindLeftMove(CandidatePoints, PointPositions);
      If NextVertex = Invalid;
          [NextVertex, Position] ← FindPointOnLine(CurrentLine, SortedColumn);
          If NextVertex = Invalid;
              [NextVertex, Position] ← FindRightMove(CandidatePoints, PointPositions);
      If NextVertex ≠ CurrentVertex;      /* NextVertex is the right neighbor of CurrentVertex*/
          CurrentVertex ← NextVertex;      CurrentLine ← RowValue(Position);
          SortedColumn ← ColumnValue(Position);      VertexList.add(CurrentVertex);
      Else
          VertexList ← VertexList − CurrentVertex;      OwnIdentity ← (i, j);
          Successor ← FindSuccessor(position);      PredecessorList(Successor) ← OwnIdentity;
          Flag ← FALSE; /* Go out of the while loop. */
   Else
      [NextVertex, Position] ← GetPreviousVertex(CurrentLine, SortedColumn);
      SortedColumn ← ColumnValue(Position);      VertexList.add(CurrentVertex);
End While;    /* The list of index values of all processors in a chain are formed below */
temp ← Successor;
While temp <> OwnIdentity do
    IndexList.add(temp);      temp ← PredecessorList(temp);
End While;    /* The processor with smallest index value now reports the polygon */
SmallestIdentity ← min(IndexList);
If SmallestIdentity = OwnIdentity
    Concatenate the vertexLists of all processors in IndexList;      Report polygon;
End DetectPolygon
```

Fig. 8. The DetectPolygon procedure for general basis polygons.

$\lceil \frac{n}{k} \rceil \times \lceil \frac{n}{k} \rceil$. Each of the submatrices is then directly mapped onto corresponding processor in the processor array, leading to the above result for task arrangement. Therefore, the upper limit of the time complexity is $O\left(\lceil \frac{n}{k} \rceil^2 n\right)$. Thus, the algorithms are upward scalable with increasing number of lines for a given processor array by a factor of $\lceil \frac{n}{k} \rceil^2$.

3.2.3. Issues in implementation and testing

In computational geometry many efficient algorithms, when implemented naively, result in fragile computer programs due to numerical errors introduced by finite precision of the floating point arithmetic [17–19]. In the implementation of our algorithm we have taken a few measures to circumvent this problem. They are as follows:

1. The basic quantities involved in the computation are the coordinate values of various points, which are in decimal form. During the computation they are changed into rational fractions of the form $x_i = \frac{p_i}{q_i}$ where p_i and q_i are integers.
2. All arithmetic computations involving solution of two simultaneous linear equations, angle computation, etc. are performed in integer field only using these rational fractions and leaving the results also as rational fractions. Thus no division by zero is involved.
3. Mutual comparison of the computed results with each other, or comparison with some constants like zero, are performed within a tolerance of some small quantity, say Δ. This solves the problem of exact value matching, leaving some inaccuracy in the result, but without causing any bottleneck.

We have tested the implementation of the algorithms with several synthetically generated data sets. We have used random number generator to generate the values of the end points of the input lines. However depending on which algorithm is being tested, later some lines are manually added/deleted from these randomly generated set of lines to add/remove special features such as degenerate structure, nested polygon etc.

4. Conclusion

In this paper we have presented two algorithms for detecting basis polygons created by n straight line segments. While the first algorithm is restricted to convex polygons, the second one can handle any general situation that can arise. Both the algorithms uses n^2 processors with a total shared memory requirement of $O(n^2)$. The worst-case computational complexity is $O(n)$. Both the algorithms have been simulated and tested successfully on samples of different polygonal structures.

Although the second algorithm can detect the basis polygons of all types, it has a small side effect. For a set of connected basis polygons it produces an extra vertex list that describes the outer periphery of the connected basis polygons put together. This can easily be ascertained from the fact that at the leftmost intersection point at least

two processors start traversals along different lines. One of these traversals starts along the line making the smallest angle with y axis and it will always trace the periphery of the connected basis polygons (e.g., $<acl_3cdr_4dr_2db\ r_1br_3bl_4bal_2al_1a>$ in Fig. 2). This polygon can be detected and rejected easily in a post-processing stage, or its computation can be aborted altogether at the beginning of the DetectPolygon by inserting suitable test conditions.

References

[1] P.V.C. Hough, Methods and means for recognizing complex patterns, US Patent 3069654, 1962.

[2] P.O. Duda, P.E. Hart, Use of the Hough transformation to detect lines and curves in pictures, Communications of the ACM 15 (1) (1972) 11–15.

[3] C. Guerra, S. Hambrusch, Parallel algorithms for line detection in a mesh, Journal of Parallel and Distributed Computing 6 (February) (1989) 1–19.

[4] R.E. Cypher, J.L.C. Sanz, L. Snyder, The Hough transform has $O(N)$ complexity on $N \times N$ mesh connected computers, SIAM Journal of Computing 19 (October) (1990) 805–820.

[5] P. Yi, H.Y.H. Chuang, Parallel Hough transform algorithms on SIMD hypercube array, in: Proceedings of the International Conference on Parallel Processing, August 1990, pp. 83–86.

[6] T. Asano, K. Obokata, T. Tokuyama, On detecting digital line components in a binary image, Proceedings of the Workshop on Computational Geometry, Calcutta, India, March 18–19, 2002.

[7] A. Sen, M. De, B.P. Sinha, A. Mukherjee, A new parallel algorithm for identification of straight lines in images, Proc. 8th International Conference on Advanced Computing and Communications, December 14–16, 2000, pp. 152–159.

[8] D. McCallum, D. Avis, A linear algorithm for finding the convex hull of a simple polygon, Information Processing Letters 9 (1979) 201–206.

[9] A.A. Melkman, On-line construction of the convex hull of a simple polyline, Information Processing Letters 25 (1987) 11–12.

[10] P. Heckbert (Ed.), Graphics Gems IV, Academic Press, New York, 1994.

[11] G.T. Toussaint, A simple linear algorithm for intersecting convex polygons, The Visual Computer 1 (1985) 118–123.

[12] J. O'Rourke, A new linear algorithm for intersecting convex polygons, Computer Graphics Image Processing 19 (1982) 384–391.

[13] S. Kundu, A new $O(n\log n)$ algorithm for computing the intersection of convex polygons, Pattern Recognition 20 (1987) 419–424.

[14] G.T. Toussaint, An optimal algorithm for computing the minimum vertex distance between two crossing convex polygons, Computing 32 (1984) 357–364.

[15] S. Sen Gupta, B.P. Sinha, An $O(\log n)$ time algorithm for testing isomorphism of maximal outerplanar graphs, Journal of Parallel and Distributed Computing 56 (1999) 144–156.

[16] G.T. Toussaint, On a convex hull algorithm for polygons and its application to triangulation problems, Pattern Recognition 15 (1982) 23–29.

[17] C.M. Hoffmann, The problem of accuracy and robustness in geometrical calculation, IEEE Computer 22 (3) (1989) 31–41.

[18] K. Sugihara, How to make geometrical algorithms robust, IEICE Transactions of Information and System E83-D (3) (2000) 447–454.

[19] D. Fogaras, K. Sugihara, Topology-orineted construction of line arrangements, IEICE Transactions of Fundamentals E85-A (4) (2002) 930–937.